Introduction to Assembly Programming language

Introduction to Assembly Programming language

- ***Machine Language***
  - ❖ Programs consist of 0s and 1s are called machine language.
  - ❖ ***Assembly Language*** provides ***mnemonics*** for machine code instructions.
  - ❖ ***Mnemonics*** refer to codes and abbreviations to make it easier for the users to remember.

- ***Low/High-level languages***
  - ❖ Assembly Language is a low-level language. Deals directly with the internal structure of CPU. ***Assembler*** translates Assembly language program into machine code.
  - ❖ In high-level languages, Pascal, Basic, C; the programmer does not have to be concerned with internal details of the CPU. ***Compilers*** translate the program into machine code.

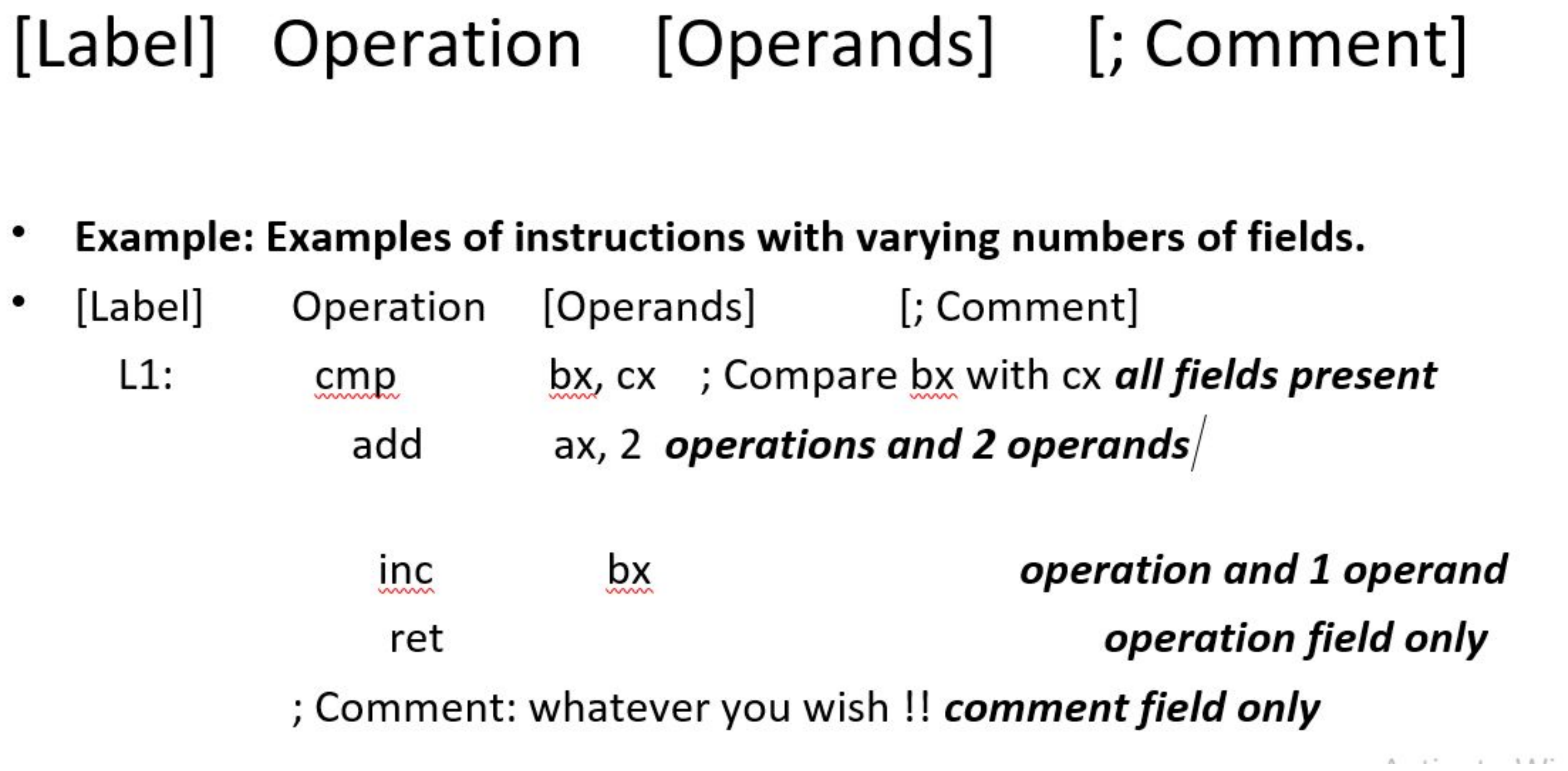
Introduction to Assembly Programming language

- ***Machine Language***
  - Programs consist of 0s and 1s are called machine language.
  - Assembly Language program consists of series of lines of Assembly language ***instructions***.
  - ***Instructions*** consist of a ***mnemonic*** and ***operand(s).***

- ***MOV instruction***

$$\underbrace{\textbf{MOV}}_{\textit{mnemonic}} \quad \underbrace{\textbf{destination, source}}_{\textit{operands}}; \qquad \textit{copy source operand to destination}$$

Format of Assembly Language Instructions



[Label]   Operation   [Operands]   [; Comment]

- **Example: Examples of instructions with varying numbers of fields.**
- [Label]   Operation   [Operands]   [; Comment]

L1:   cmp   bx, cx   ; Compare bx with cx ***all fields present***

add   ax, 2 ***operations and 2 operands***

inc   bx   ***operation and 1 operand***

ret   ***operation field only***

; Comment: whatever you wish !! ***comment field only***

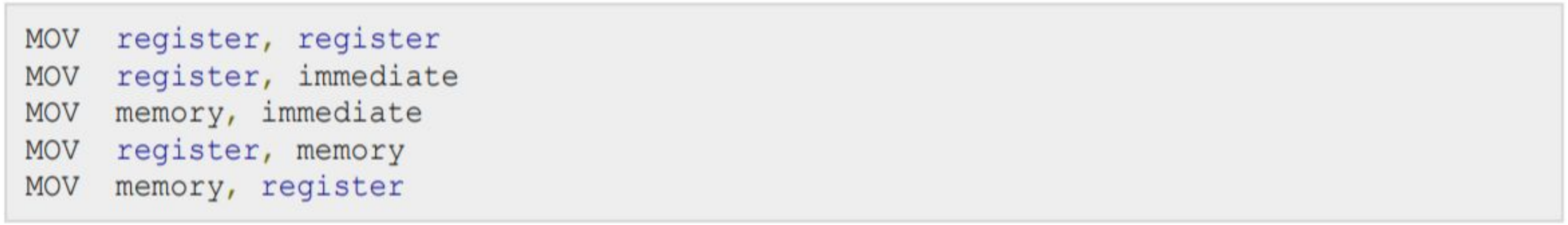
MOV Instruction

<u>The MOV Instruction:</u>

We have already used the MOV instruction that is used for moving data from one storage space to another.
The MOV instruction takes two operands.

<u>SYNTAX:</u>

MOV destination, source
The MOV instruction may have one of the following five forms:

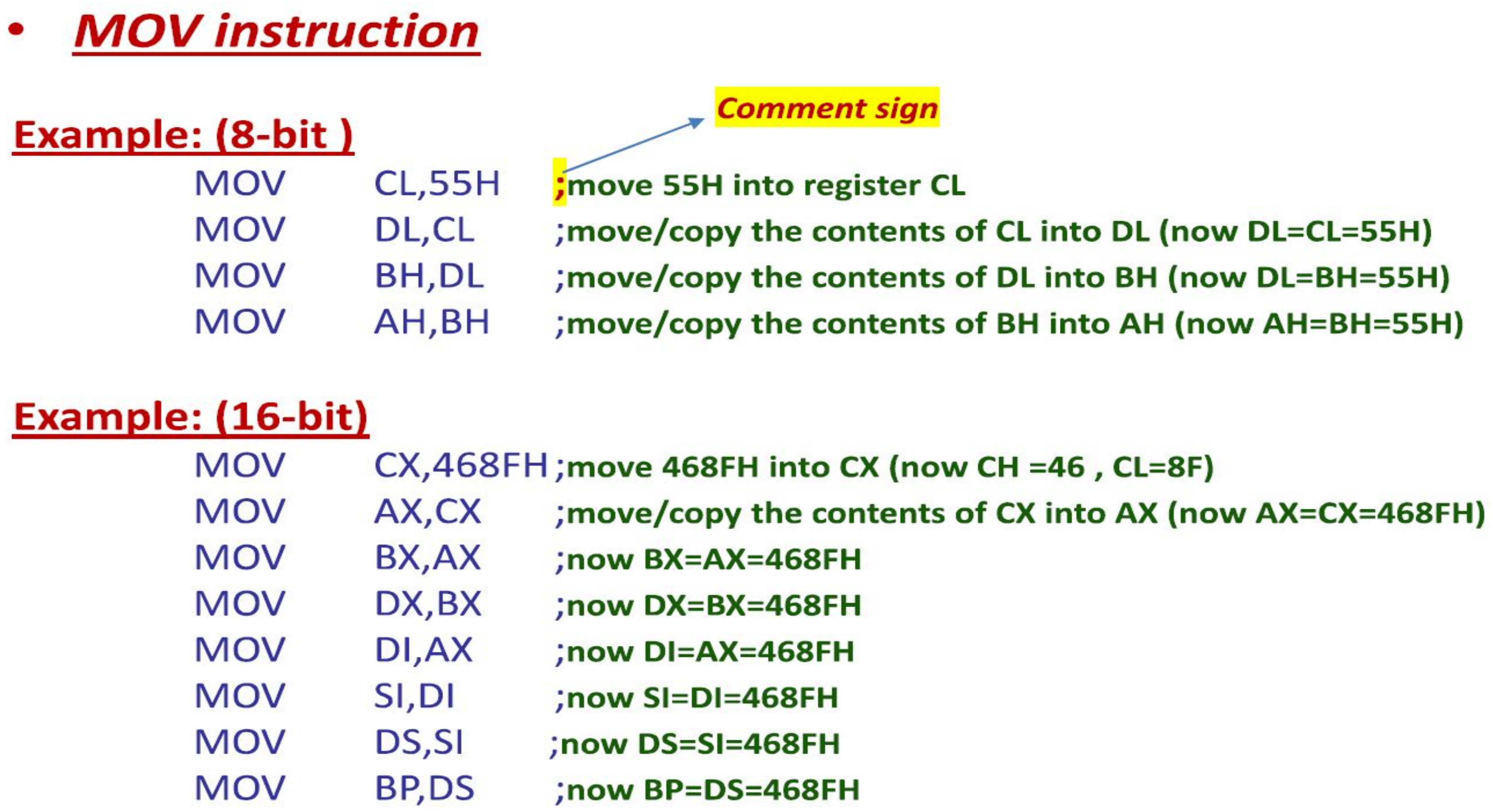


```
MOV  register, register
MOV  register, immediate
MOV  memory, immediate
MOV  register, memory
MOV  memory, register
```

Please note that:

- Both the operands in MOV operation should be of same size
- The value of source operand remains unchanged

MOV Instruction



- ***MOV instruction***

Comment sign

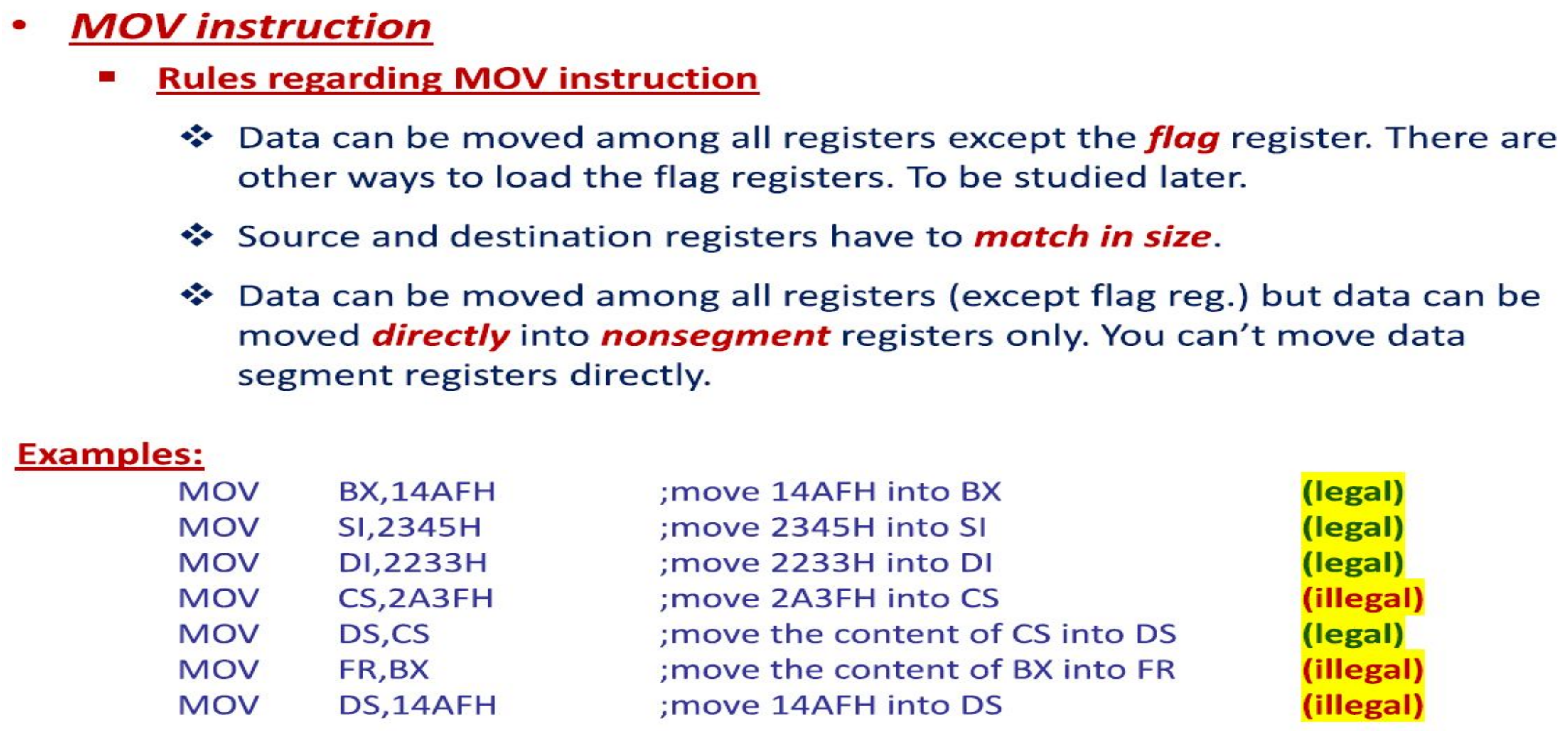
Example: (8-bit)

```
MOV     CL,55H     ;move 55H into register CL
MOV     DL,CL      ;move/copy the contents of CL into DL (now DL=CL=55H)
MOV     BH,DL      ;move/copy the contents of DL into BH (now DL=BH=55H)
MOV     AH,BH      ;move/copy the contents of BH into AH (now AH=BH=55H)
```

Example: (16-bit)

```
MOV     CX,468FH ;move 468FH into CX (now CH =46 , CL=8F)
MOV     AX,CX      ;move/copy the contents of CX into AX (now AX=CX=468FH)
MOV     BX,AX      ;now BX=AX=468FH
MOV     DX,BX      ;now DX=BX=468FH
MOV     DI,AX      ;now DI=AX=468FH
MOV     SI,DI      ;now SI=DI=468FH
MOV     DS,SI     ;now DS=SI=468FH
MOV     BP,DS      ;now BP=DS=468FH
```

MOV Instruction



- ***MOV instruction***
  - **Rules regarding MOV instruction**
    - Data can be moved among all registers except the ***flag*** register. There are other ways to load the flag registers. To be studied later.
    - Source and destination registers have to ***match in size***.
    - Data can be moved among all registers (except flag reg.) but data can be moved ***directly*** into ***nonsegment*** registers only. You can't move data segment registers directly.

Examples:

```
MOV    BX,14AFH     ;move 14AFH into BX                  (legal)
MOV    SI,2345H     ;move 2345H into SI                  (legal)
MOV    DI,2233H     ;move 2233H into DI                  (legal)
MOV    CS,2A3FH     ;move 2A3FH into CS                  (illegal)
MOV    DS,CS        ;move the content of CS into DS      (legal)
MOV    FR,BX        ;move the content of BX into FR      (illegal)
MOV    DS,14AFH     ;move 14AFH into DS                  (illegal)
```

MOV Instruction

- ***MOV instruction***
  - **Important points**
    - Data values cannot be loaded directly into (***CS***,***DS***,***SS*** and ***ES***)

```
MOV     AX,1234H          ; load 1234H into AX
MOV     SS,AX             ; load the value in AX into SS
```

    - Sizes of the values:

```
MOV     BX,2H             ; BX=0002H,        BL:02H,  BH:00H

MOV     AL,123H           ; illegal (larger than 1 byte)
MOV     AX,3AFF21H        ; illegal (larger than 2 bytes)
```

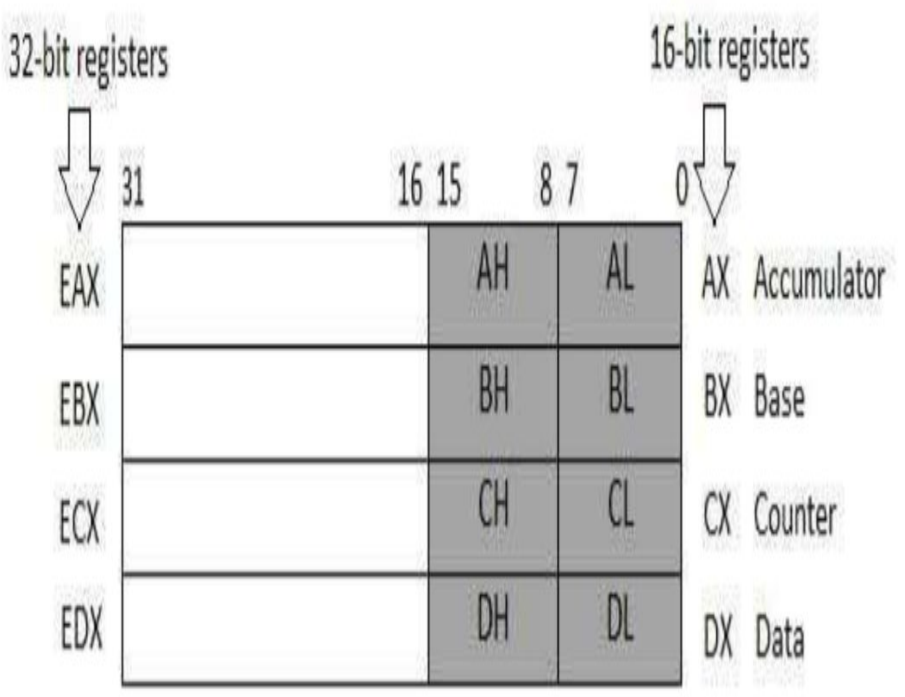
Assembly Registers

- To speed up the processor operations, the processor includes some internal memory storage locations, called registers. The registers stores data elements for processing without having to access the memory. A limited number of registers are built into the processor chip.
- The registers are grouped into three categories:
    - General registers
    - Control registers
    - Segment registers
- The general registers are further divided into the following groups:
    - Data registers
    - Pointer registers
    - Index registers

Assembly Registers (Data Register)

Some of these data registers has specific used in arithmetical operations.

- AX is the primary accumulator; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX, or AX or AL register according to the size of the operand.
- BX is known as the base register as it could be used in indexed addressing.
- CX is known as the count register as the ECX, CX registers store the loop count in iterative operations.
- DX is known as the data register. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.



Assembly Variables

- In Assembly programming, the variable are all defined by bytes only.
  - **DB –** Define Byte (Size – 1 Byte)
  - **DW –** Define Word (Size – 2 Byte)
  - **DD –** Define Double word (Size – 4 Bytes)
  - **DQ –** Define Quad word (Size – 8 Bytes)
  - **DT –** Define Ten Bytes (Size – 10 Bytes)

Arithmetic Instructions

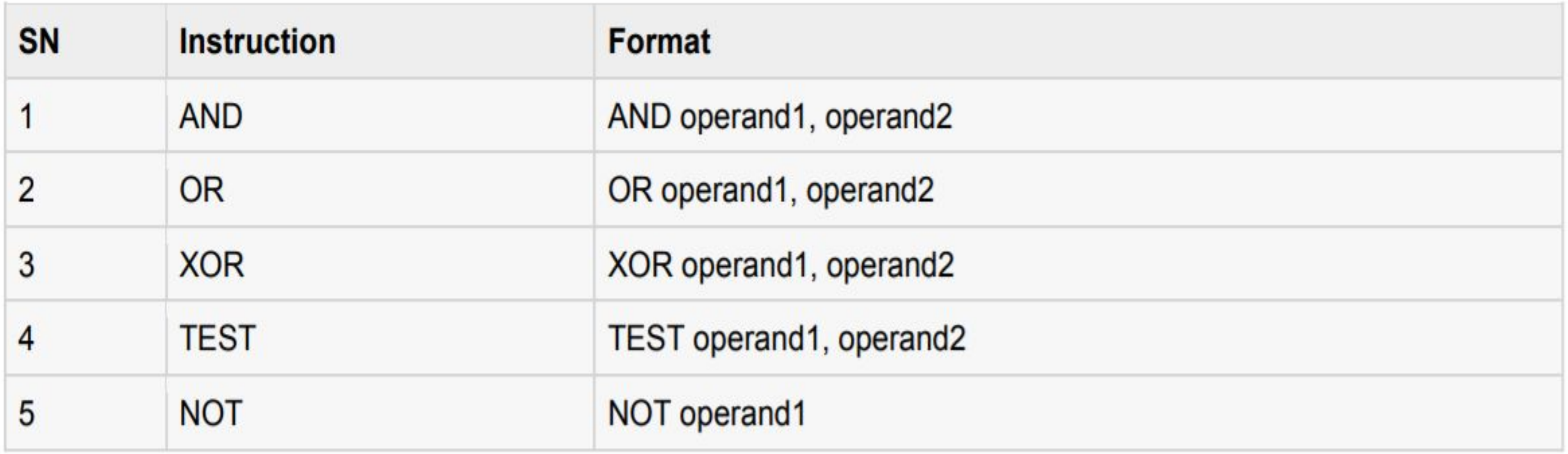
- Arithmetic and logic instructions can be performed on 8-bit (byte) and 16-bit values. The first operand has to be a register and the result is stored in that register.

Arithmetic Instructions

- **Increment** the contents of BX register by 4
  - ADD BX, 4
- **Add** the contents of AX register with the contents of CX register
  - ADD AX, CX
- **Subtract** 1 from the contents of AL register
  - SUB AL, 1
- **Subtract** the contents of CX register from the contents of DX register
  - SUB DX, CX
- **Multiply** AL by BL, the result will be in AX
  - MUL BL
- **Divide** the contents of AX register with the value of CL and store the result in AX
  - DIV CL
- **Increase** or Decrease the contents of BX register by 1
  - **INC** BX ; Increase
  - **DEC** BX ; Decrease
- **Clear** the contents of AX register
  - XOR AX, AX
- **Negation** of a register value
  - NEG AX

Logical Instructions

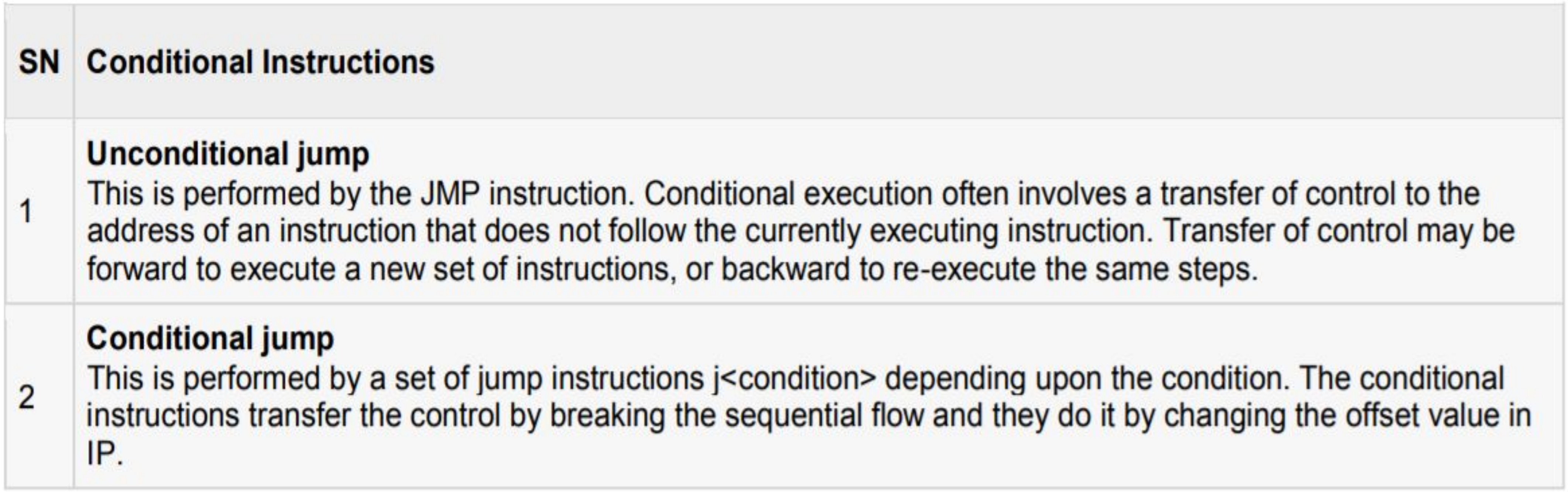
- The format for these instructions:



SN	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

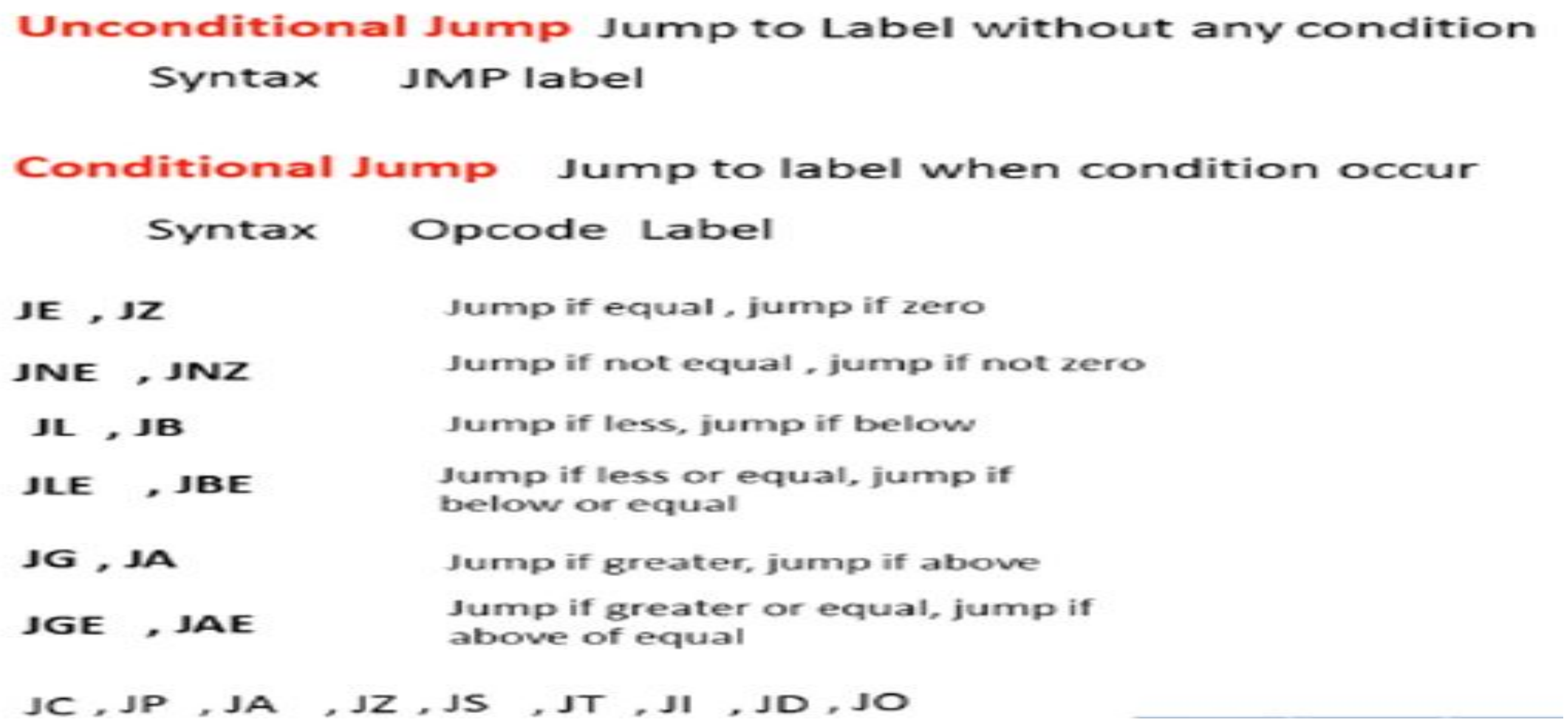
Assembly Conditions

- Conditional execution in assembly language is accomplished by several looping and branching instructions. These instructions can change the flow of control in a program. Conditional execution is observed in two scenarios:



SN	Conditional Instructions
1	**Unconditional jump** This is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward to execute a new set of instructions, or backward to re-execute the same steps.
2	**Conditional jump** This is performed by a set of jump instructions j<condition> depending upon the condition. The conditional instructions transfer the control by breaking the sequential flow and they do it by changing the offset value in IP.

Assembly Conditions



Unconditional Jump Jump to Label without any condition

Syntax JMP label

Conditional Jump Jump to label when condition occur

Syntax Opcode Label

JE , JZ	Jump if equal , jump if zero
JNE , JNZ	Jump if not equal , jump if not zero
JL , JB	Jump if less, jump if below
JLE , JBE	Jump if less or equal, jump if below or equal
JG , JA	Jump if greater, jump if above
JGE , JAE	Jump if greater or equal, jump if above of equal

JC , JP , JA , JZ , JS , JT , JI , JD , JO

Assembly Loops

- The LOOP instruction is a combination of a decrement of CX and a conditional jump. In the 8086, LOOP decrements CX and if CX is not equal to zero, it jumps to the address indicated by the label. If CX becomes a 0, the next sequential instruction executes.
- **Syntax:**

```
Labelname:
.
.
.
Loop labelname
```

Assembly Procedure

- A procedure is a collection of instructions to which we can direct the flow of our program, and once the execution of these instructions is over control is given back to the next line to process of the code which called on the procedure.
- At the time of invoking a procedure the address of the next instruction of the program is kept on the stack so that, once the flow of the program has been transferred and the procedure is done, one can return to the next line of the original program, the one which called the procedure.

Syntax:

The syntax for procedure declaration:

```
name PROC

      ; here goes the code
      ; of the procedure ...

RET
name ENDP
```

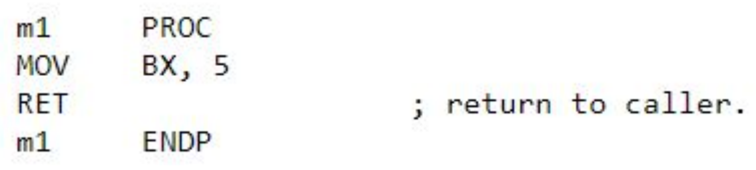
name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Probably, **RET** instruction is used to return to operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

PROC and **ENDP** are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

CALL instruction is used to call a procedure.

Here is an example:

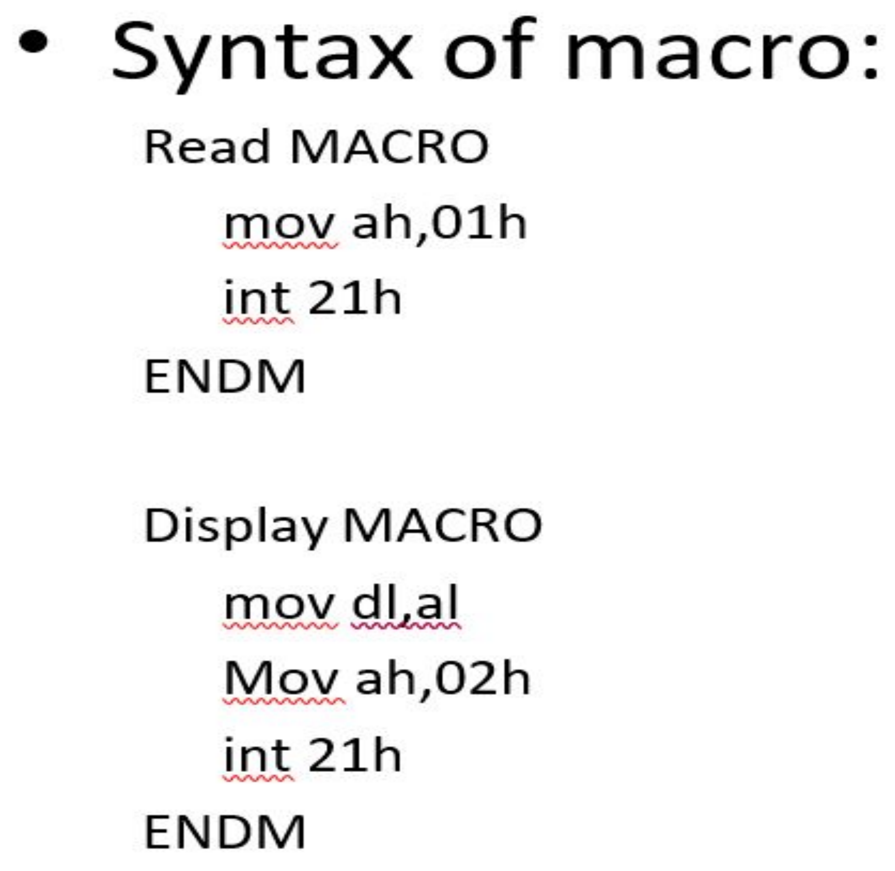


```
m1      PROC
MOV     BX, 5
RET                         ; return to caller.
m1      ENDP
```

Assembly MACRO

- A macro is a group of repetitive instructions in a program which are codified only once and can be used as many times as necessary.
- A macro can be defined anywhere in program using the directives MACRO and ENDM

Syntax:



- Syntax of macro:

```
Read MACRO
    mov ah,01h
    int 21h
ENDM

Display MACRO
    mov dl,al
    Mov ah,02h
    int 21h
ENDM
```

Passing parameters to a macro

```
•   Display MACRO MSG
            mov dx, offset msg
            mov ah,09h
            int 21h
  ENDM
```

Here parameter is MSG

The parameter MSG can be replaced by msg1 or msg2 while calling…

Calling macro:

```
            DISPLAY MSG1

            DISPLAY MSG2
```

```
MSG1 db “I am Fine $”
 MSG2 db “Hello, How are you..? $”
```

Exercise

What errors are presenting in the following?

- mov ax 3d
- mov 23, ax
- mov cx, ch
- move ax, 1h
- add 2, cx
- add 3, 6
- inc ax, 2

Sample of Assembly Programming Exercise

- **Write an assembly program to find the area of a triangle/Circle.**
- **Write an assembly code to find an average of 3/5 numbers.**
- **Write an assembly code to print all natural numbers from 0 to 9 / from 9 to 0**
- **Write an assembly code to check whether a input number is odd or even number**

****And the problems that have been solved in the lab class.**